1

Introduction

> *"Computer Science is no more about computers than astronomy is about telescopes"*, attributed to Edsger Dijkstra. [1]

> *"Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry."* , Paul Graham 2003. [2]

> *"The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block."*, Alan Cobham, 1964

The origin of much of science and medicine can be traced back to the ancient Babylonians. But perhaps their greatest contribution to humanity was the invention of the *place-value number system*. This is the idea that we can represent any number using a fixed number of digits, whereby the *position* of the digit is used to determine the corresponding value, as opposed to system such as Roman numerals, where every symbol has a fixed numerical value regardless of position. For example, the distance to the moon is 238,900 of our miles or 259,956 Roman miles. The latter quantity, expressed in standard Roman numerals is

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

[1] This quote is typically read as disparaging the importance of actual physical computers in Computer Science, but note that telescopes are absolutely essential to astronomy and are our only means of connecting theoretical speculations with actual experimental observations.

[2] To be fair, in the following sentence Graham says "you need to know how to calculate time and space complexity and about Turing completeness". Apparently, NP-hardness, randomization, cryptography, and quantum computing are not essential to a hacker's education.

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

MMMMMMMMMMMMMMMDCCCCLVI
```

Writing the distance to the sun in Roman numerals would require about 100,000 symbols: a 50 page book just containing this single number!

This means that for someone that thinks of numbers in an additive system like Roman numerals, quantities like the distance to the moon or sun are not merely large- they are *unspeakable*: cannot be expressed or even grasped. It's no wonder that Eratosthene, who was the first person to calculate the earth's diameter (up to about ten percent error) and Hipparchus who was the first to calculate the distance to the moon, did not use a Roman-numeral type system but rather the Babylonian sexadecimal (i.e., base 60) place-value system.

The Babylonians also invented the precursors of "standard algorithms" that we were all taught in elementary school for adding and multiplying numbers.[3] These algorithms and their variants have been of course essential to people throughout history working with abaci, papyrus, or pencil and paper, but in our computer age, do they really serve any purpose beyond torturing third graders?

[3] For more on the actual algorithms the Babylonians used, see Knuth's paper and Neugebauer's classic book.

To answer this question, let us try to see in what sense is the standard digit by digit multiplication algorithm "better" than the straightforward implementation of multiplication as iterated addition. Let's start by more formally describing both algorithms:

Naive multiplication algorithm:
Input: Non-negative integers x, y
Operation:
1. Let $result \leftarrow 0$.
2. For $i = 1, \ldots, y$: set $result \leftarrow result + x$
3. Output $result$

Standard gradeschool multiplication algorithm:
Input: Non-negative integers x, y
Operation:
1. Let n be number of digits of y, and set $result \leftarrow 0$.
2. For $i = 0, \ldots, n-1$: set $result \leftarrow result + 10^i \times$

$y_i \times x$, where y_i is the i-th digit of y (i.e. $y = 10^0 y_0 + 10^1 y_1 + \cdots + y_{n-1} 10^{n-1}$)
3. Output *result*

Both algorithms assume that we already know how to add numbers, and the second one also assumes that we can multiply a number by a power of 10 (which is after all a simple shift) as well as multiply by a single digit (which like addition, is done by multiplying each digit and propagating carries). Now suppose that x and y are two numbers of n decimal digits each. Adding two such numbers takes at least n single digit additions (depending on how many times we need to use a "carry"), and so adding x to itself y times will take at least $n \cdot y$ single digit additions. In contrast, the standard gradeschool algorithm reduces this problem to taking n products of x with a single digit (which require up to $2n$ single digit operations each, depending on carries) and then adding all of those together (total of n additions, which, again depending on carries, would cost at most $2n^2$ single digit operations) for a total of at most $4n^2$ single digit operations. How much faster would $4n^2$ operations be than $n \cdot y$? and would this make any difference in a modern computer?

Let us consider the case of multiplying 64 bit or 20 digit numbers.[4] That is, the task of multiplying two numbers x, y that are between 10^{19} and 10^{20}. Since in this case $n = 20$, the standard algorithm would use at most $4n^2 = 1600$ single digit operations, while repeated addition would require at least $n \cdot y \geq 20 \cdot 10^{19}$ single digit operations. To understand the difference, consider that a human being might do a single digit operation in about 2 seconds, requiring just under an hour to complete the calculation of $x \times y$ using the gradeschool algorithm. In contrast, even though it is more than a billion times faster, a modern PC that computes $x \times y$ using naïve iterated addition would require about $10^{20}/10^9 = 10^{11}$ seconds (which is more than three millenia!) to compute the same result.

We see that computers have not made algorithms obsolete. On the contrary, the vast increase in our ability to measure, store, and communicate data has led to a much higher demand on developing better and more sophisticated algorithms that can allow us to make better decisions based on these data. We also see that to a large extent the notion of *algorithm* is independent of the actual computing device that will execute it. The digit-by-digit standard algorithm is vastly better than iterated addition regardless if the technology implementing it is a silicon based chip or a third grader with pen and paper.

[4] This is a common size in several programming languages; for example the `long` data type in the Java programming language, and (depending on architecture) the `long` or `long long` types in C.

Theoretical computer science is largely about studying the *inherent* properties of algorithms and computation, that are independent of current technology. We ask questions that already pondered by the Babylonians, such as "what is the best way to multiply two numbers?" as well as those that rely on cutting-edge science such as "could we use the effects of quantum entanglement to factor numbers faster" and many in between. These types of questions are the topic of this course.

In Computer Science parlance, a scheme such as the decimal (or sexadecimal) positional representation for numbers is known as a *data structure*, while the operations on this representations are known as *algorithms*. Data structures and algorithms have enabled amazing applications, but their importance goes beyond their practical utility. Structures from computer science, such as bits, strings, graphs, and even the notion of a program itself, as well as concepts such as universality and replication, have not just found (many) practical uses but contributed a new language and a new way to view the world.

1.0.1 Example: A faster way to multiply

Once you think of the standard digit-by-digit multiplication algorithm, it seems like obviously the "right" way to multiply numbers. Indeed, in 1960, the famous mathematician Andrey Kolmogorov organized a seminar at Moscow State University in which he conjectured that every algorithm for multiplying two n digit numbers would require a number of basic operations that is proportional to n^2.[5] Another way to say it, is that he conjectured that in any multiplication algorithm, doubling the number of digits would *quadruple* the number of basic operations required.

[5] That is at least some n^2/C operations for some constant C or, using "Big Oh notation", $\Omega(n^2)$ operations. See the *mathematical background* section for a precise definition of big Oh notation.

A young student named Anatoly Karatsuba was in the audience, and within a week he found an algorithm that requires only about $Cn^{1.6}$ operations for some constant C. Such a number becomes much smaller than n^2 as n grows.[6] Amazingly, Karatsuba's algorithm is based on a faster way to multiply *two digit* numbers.

[6] At the time of this writing, the standard Python multiplication implementation switches from the elementary school algorithm to Karatsuba's algorithm when multiplying numbers larger than 1000 bits long.

Suppose that $x, y \in [100] = \{0, \ldots, 99\}$ are a pair of two-digit numbers. Let's write $\overline{x}$ for the "tens" digit of x, and $\underline{x}$ for the "ones" digit, so that $x = 10\overline{x} + \underline{x}$, and write similarly $y = 10\overline{y} + \underline{y}$ for $\overline{y}, \underline{y} \in [10]$. The gradeschool algorithm for multiplying x and y is illustrated in Fig. 1.1.

The gradeschool algorithm works by transforming the task of

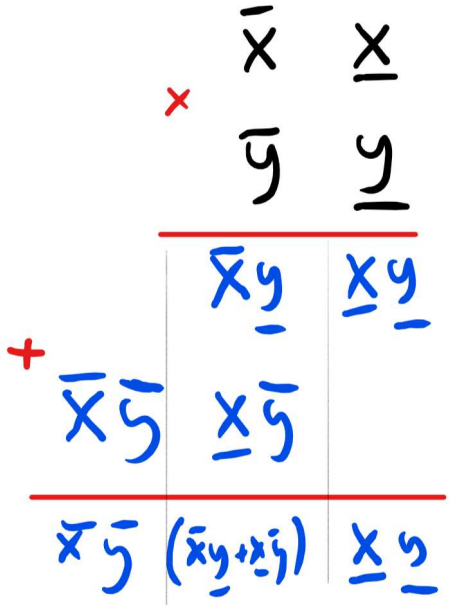

Figure 1.1: The gradeschool multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. It uses the formula $(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y}$.

multiplying a pair of two digit number into *four* single-digit multiplications via the formula

$$(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y} \tag{1.1}$$

Karatsuba's algorithm is based on the observation that we can express this also as

$$(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10\left[(\overline{x} + \underline{x})(\overline{y} + \underline{y})\right] - 10\overline{x}\overline{y} - (10+1)\underline{x}\underline{y} \tag{1.2}$$

which reduces multiplying the two-digit number x and y to computing the following three "simple" products: $\overline{x}\overline{y}$, $\underline{x}\underline{y}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$.[7]

Of course if all we wanted to was to multiply two digit numbers, we wouldn't really need any clever algorithms. It turns out that we can repeatedly apply the same idea, and use them to multiply 4-digit numbers, 8-digit numbers, 16-digit numbers, and so on and so forth. If we used the gradeschool based approach then our cost for doubling the number of digits would be to *quadruple* the number

[7] The last term is not exactly a single digit multiplication as $\overline{x} + \underline{x}$ and $\overline{y} + \underline{y}$ are numbers between 0 and 18 and not between 0 and 9. As we'll see, it turns out that does not make much of a difference, since when we use the algorithm recursively, this term will have essentially half the number of digits as the original input.

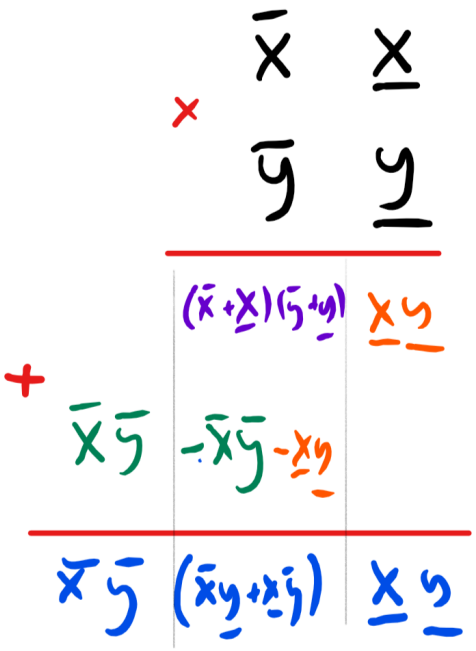

Figure 1.2: Karatsuba's multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. We compute the three orange, green and purple products $\underline{x}\underline{y}$, $\overline{x}\overline{y}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$ and then add and subtract them to obtain the result.

of multiplications, which for $n = 2^\ell$ digits would result in about $4^\ell = n^2$ operations. In contrast, in Karatsuba's approach doubling the number of digits only *triples* the number of operations, which means that for $n = 2^\ell$ digits we require about $3^\ell = n^{\log_2 3} \sim n^{1.58}$ operations.

Specifically, we use a *recursive* strategy as follows:

Karatsuba Multiplication:
Input: Non negative integers x, y each of at most n digits
Operation:
1. If $n \leq 2$ then return $x \cdot y$ (using a constant number of single digit multiplications) 2. Otherwise, let $m = \lfloor n/2 \rfloor$, and write $x = 10^m\overline{x} + \underline{x}$ and $y = 10^m\overline{y} + \underline{y}$. [8]
2. Use *recursion* to compute $A = \overline{x}\overline{y}$, $B = \underline{y}\underline{y}$ and $C = (\overline{x} + \underline{x})(\overline{y} + \underline{y})$. Note that all the numbers will have at most $m + 1$ digits.
3. Return $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$

To understand why the output will be correct, first note that for $n > 2$, it will always hold that $m < n - 1$, and hence the recursive calls will always be for multiplying numbers with a smaller number

[8] Recall that for a number x, $\lfloor x \rfloor$ is obtained by "rounding down" x to the largest integer smaller or equal to x.

of digits, and (since eventually we will get to single or double digit numbers) the algorithm will indeed terminate. Now, since $x = 10^m\overline{x} + \underline{x}$ and $y = 10^m\overline{y} + \underline{y}$,

$$x \times y = 10^n\overline{x} \cdot \overline{y} + 10^m(\overline{xy} + \underline{xy}) + \underline{xy} \tag{1.3}$$

But we can also write the same expression as

$$x \times y = 10^n\overline{x} \cdot \overline{y} + 10^m \left[(\overline{x} + \underline{x})(\overline{y} + \underline{y}) - \underline{xy} - \overline{xy}\right] + \underline{xy} \tag{1.4}$$

which equals to the value $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$ returned by the algorithm.

The key observation is that the formula Eq. (1.4) reduces computing the product of two n digit numbers to computing *three* products of $\lfloor n/2 \rfloor$ digit numbers (namely $\overline{xy}$, $\underline{yy}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$ as well as performing a constant number (in fact eight) additions, subtractions, and multiplications by 10^n or $10^{\lfloor n/2 \rfloor}$ (the latter corresponding to simple shifts). Intuitively this means that as the number of digits *doubles*, the cost of multiplying *triples* instead of quadrupling, as happens in the naive algorithm. This implies that multiplying numbers of $n = 2^\ell$ digits costs about $3^\ell = n^{\log_2 3} \sim n^{1.585}$ operations. In a Exercise 1.3, you will formally show that the number of single digit operations that Karatsuba's algorithm uses for multiplying n digit integers is at most $O(n^{\log_2 3})$ (see also Fig. 1.3).

R **Ceilings, floors, and rounding** One of the benefits of using big Oh notation is that we can allow ourselves to be a little looser with issues such as rounding numbers etc.. For example, the natural way to describe Karatsuba's algorithm's running time is as following the recursive equation $T(n) = 3T(n/2) + O(n)$ but of course if n is not even then we cannot recursively invoke the algorithm on $n/2$-digit integers. Rather, the true recursion is $T(n) = 3T(\lfloor n/2 \rfloor + 1) + O(n)$. However, this will not make much difference when we don't worry about constant factors, since its not hard to show that $T(n + O(1)) \leq T(n) + o(T(n))$ for the functions we care about (which turns out to be enough to carry over the same recursion). Another way to show that this doesn't hurt us is to note that for every number n, we can find $n' \leq 2n$ such that n' is a power of two. Thus we can always "pad" the input by adding some input bits to make sure the number of digits is a power of two, in which case we will never run into these rounding issues. These kind of tricks work

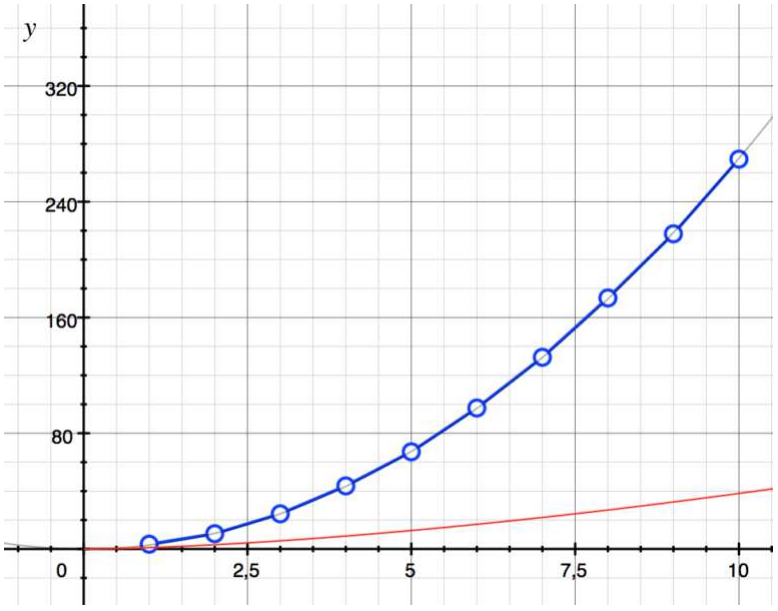

Figure 1.3: Running time of Karatsuba's algorithm vs. the Gradeschool algorithm. Figure by Marina Mele.

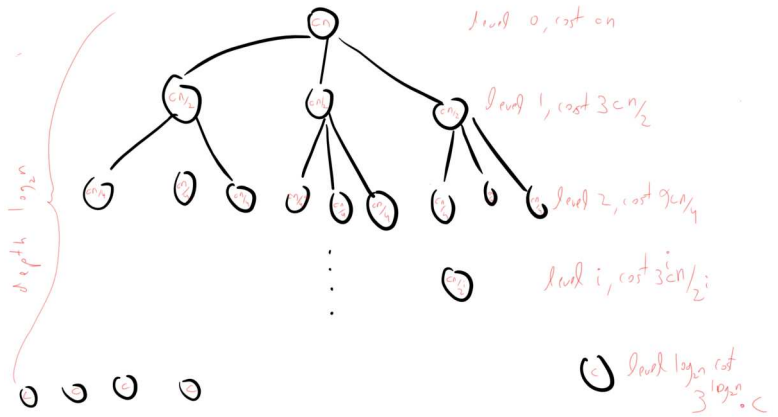

Figure 1.4: Karatsuba's algorithm reduces an n-bit multiplication to three $n/2$-bit multiplications, which in turn are reduced to nine $n/4$-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth $\log_2 n$, where at the root the extra cost is cn operations, at the first level the extra cost is $c(n/2)$ operations, and at each of the 3^i nodes of level i, the extra cost is $c(n/2^i)$. The total cost is $cn\sum_{i=0}^{\log_2 n}(3/2)^i \leq 2cn^{\log_2 3}$ by the formula for summing a geometric series.

not just in the context of multiplication algorithms but in many other cases as well. Thus most of the time we can safely ignore these kind of "rounding issues".

1.0.2 *Beyond Karatsuba's algorithm*

It turns out that the ideas of Karatsuba can be further extended to yield asymptotically faster multiplication algorithms, as was shown by Toom and Cook in the 1960s. But this was not the end of the line. In 1971, Schönhage and Strassen gave an even faster algorithm using the *Fast Fourier Transform*; their idea was to somehow treat integers as "signals" and do the multiplication more efficiently by moving to the Fourier domain.[9] The latest asymptotic improvement was given by Fürer in 2007 (though it only starts beating the Schönhage-Strassen algorithm for truly astronomical numbers). And yet, despite all this progress, we still don't know whether or not there is an $O(n)$ time algorithm for multiplying two n digit numbers!

[9] The *Fourier transform* is a central tool in mathematics and engineering, used in a great number of applications. If you have not seen it yet, you will hopefully encounter it at some point in your studies.

1.0.3 *Advanced note: matrix multiplication*

(We will have several such "advanced" notes and sections throughout these lectures notes. These may assume background that not every student has, and in any case can be safely skipped over as none of the future parts will depend on them.)

It turns out that a similar idea as Karatsuba's can be used to speed up *matrix* multiplications as well. Matrices are a powerful way to represent linear equations and operations, widely used in a great many applications of scientific computing, graphics, machine learning, and many many more. One of the basic operations one can do with two matrices is to *multiply* them. For example, if $x = \begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}$ and $y = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \end{pmatrix}$ then the product of x and y is the matrix $\begin{pmatrix} x_{0,0}y_{0,0}+x_{0,1}y_{1,01} & x_{0,0}y_{1,0}+x_{0,1}y_{1,1} \\ x_{1,0}y_{0,0}+x_{1,1}y_{1,0} & x_{1,0}y_{0,1}+x_{1,1}y_{1,1} \end{pmatrix}$. You can see that we can compute this matrix by *eight* products of numbers. Now suppose that n is even and x and y are a pair of $n \times n$ matrices which we can think of as each composed of four $(n/2) \times (n/2)$ blocks $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ and $y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1}$. Then the formula for the matrix product of x and y can be expressed in the same way as above, just replacing products $x_{a,b}y_{c,d}$ with *matrix* products, and addition with matrix addition. This means that we can use the formula above to give an algorithm that *doubles* the dimension of the matrices at the expense of increasing

the number of operation by a factor of 8, which for $n = 2^\ell$ will result in $8^\ell = n^3$ operations. > In 1969 Volker Strassen noted that we can compute the product of a pair of two by two matrices using only *seven* products of numbers by observing that each entry of the matrix xy can be computed by adding and subtracting the following seven terms: $t_1 = (x_{0,0} + x_{1,1})(y_{0,0} + y_{1,1})$, $t_2 = (x_{0,0} + x_{1,1})y_{0,0}$, $t_3 = x_{0,0}(y_{0,1} - y_{1,1})$, $t_4 = x_{1,1}(y_{0,1} - y_{0,0})$, $t_5 = (x_{0,0} + x_{0,1})y_{1,1}$, $t_6 = (x_{1,0} - x_{0,0})(y_{0,0} + y_{0,1})$, $t_7 = (x_{0,1} - x_{1,1})(y_{1,0} + y_{1,1})$. Indeed, one can verify that $xy = \begin{pmatrix} t_1+t_4-t_5+t_7 & t_3+t_5 \\ t_2+t_4 & t_1+t_3-t_2+t_6 \end{pmatrix}$. This implies an algorithm with factor 7 increased cost for doubling the dimension, which means that for $n = 2^\ell$ the cost is $7^\ell = n^{\log_2 7} \sim n^{2.807}$. A long sequence of works has since improved this algorithm, and the current record has running time about $O(n^{2.373})$. Unlike the case of integer multiplication, at the moment we don't know of a nearly linear in the matrix size (i.e., an $O(n^2 polylog(n))$ time) algorithm for matrix multiplication. People have tried to use group representations, which can be thought of as generalizations of the Fourier transform, to obtain faster algorithms, but this effort has not yet succeeded.

1.0.4 Algorithms beyond arithmetic

The quest for better algorithms is by no means restricted to arithmetical tasks such as adding, multiplying or solving equations. Many *graph algorithms*, including algorithms for finding paths, matchings, spanning tress, cuts, and flows, have been discovered in the last several decades, and this is still an intensive area of research. (For example, the last few years saw many advances in algorithms for the *maximum flow* problem, borne out of surprising connections with electrical circuits and linear equation solvers.)

These algorithms are being used not just for the "natural" applications of routing network traffic or GPS-based navigation, but also for applications as varied as drug discovery through searching for structures in gene-interaction graphs to computing risks from correlations in financial investments.

Google was founded based on the *PageRank* algorithm, which is an efficient algorithm to approximate the "principal eigenvector" of (a dampened version of) the adjacency matrix of web graph. The *Akamai* company was founded based on a new data structure, known as *consistent hashing*, for a hash table where buckets are stored at different servers.

The *backpropagation algorithm*, that computes partial derivatives of a neural network in $O(n)$ instead of $O(n^2)$ time, underlies many of

the recent phenomenal successes of learning deep neural networks. Algorithms for solving linear equations under sparsity constraints, a concept known as *compressed sensing*, have been used to drastically reduce the amount and quality of data needed to analyze MRI images. This is absolutely crucial for MRI imaging of cancer tumors in children, where previously doctors needed to use anesthesia to suspend breath during the MRI exam, sometimes with dire consequences.

Even for classical questions, studied through the ages, new discoveries are still being made. For the basic task, already of importance to the Greeks, of discovering whether an integer is prime or composite, efficient probabilistic algorithms were only discovered in the 1970s, while the first deterministic polynomial-time algorithm was only found in 2002. For the related problem of actually finding the factors of a composite number, new algorithms were found in the 1980s, and (as we'll see later in this course) discoveries in the 1990s raised the tantalizing prospect of obtaining faster algorithms through the use of quantum mechanical effects.

Despite all this progress, there are still many more questions than answers in the world of algorithms. For almost all natural problems, we do not know whether the current algorithm is the "best", or whether a significantly better one is still waiting to be discovered. As we already saw, even for the classical problem of multiplying numbers we have not yet answered the age-old question of **"is multiplication harder than addition?"** .

But at least we now know the right way to *ask* it..

1.0.5 On the importance of negative results.

Finding better multiplication algorithms is undoubtedly a worthwhile endeavor. But why is it important to prove that such algorithms *don't* exist? What useful applications could possibly arise from an impossibility result?

One motivation is pure intellectual curiosity. After all, this is a question even Archimedes could have been excited about. Another reason to study impossibility results is that they correspond to the fundamental limits of our world or in other words to *laws of nature*. In physics, it turns out that the impossibility of building a *perpetual motion machine* corresponds to the *law of conservation of energy*. Other laws of nature also correspond to impossibility results: the impossibility of building a heat engine beating Carnot's bound corresponds to the second law of thermodynamics, while the impossibility of

faster-than-light information transmission is a cornerstone of special relativity. Within mathematics, while we all learned the solution for quadratic equations in high school, the impossibility of deneralizing this to equations of degree five or more gave birth to *group theory*. In his 300 B.C. book *The Elements*, the Greek mathematician Euclid based geometry on five "axioms" or "postulates". Ever since then people have suspected that four axioms are enough, and try to base the "parallel postulate" (roughly speaking, that every line has a unique parallel line of each distance) from the other four. It turns out that this was impossible, and the impossibility result gave rise to so called "non-Euclidean geometries", which turn out to be crucial for the theory of general relativity.[10]

[10] It is fine if you have not yet encountered many of the above. I hope however it sparks your curiosity!

In an analogous way, impossibility results for computation correspond to "computational laws of nature" that tell us about the fundamental limits of any information processing apparatus, whether based on silicon, neurons, or quantum particles.[11] Moreover, computer scientists have recently been finding creative approaches to *apply* computational limitations to achieve certain useful tasks. For example, much of modern Internet traffic is encrypted using the RSA encryption scheme, which relies on its security on the (conjectured) *non existence* of an efficient algorithm to perform the inverse operation for multiplication— namely, factor large integers. More recently, the Bitcoin system uses a digital analog of the "gold standard" where, instead of being based on a precious metal, minting new currency corresponds to "mining" solutions for computationally difficult problems.

[11] Indeed, some exciting recent research has been trying to use computational complexity to shed light on fundamental questions in physics such understanding black holes and reconciling general relativity with quantum mechanics.

1.1 Lecture summary

(*The notes for every lecture will end in such a "lecture summary" section that contains a few of the "take home messages" of the lecture. It is not meant to be a comprehensive summary of all the main points covered in the lecture.*)

- There can be several different algorithms to achieve the same computational task. Finding a faster algorithm can make a much bigger difference than better technology.
- Better algorithms and data structures don't just speed up calculations, but can yield new qualitative insights.
- One of the main topics of this course is studying the question of what is the *most efficient* algorithm for a given problem.

- To answer such a question we need to find ways to *prove lower bounds* on the computational resources needed to solve certain problems. That is, show an *impossiblity result* ruling out the existence of "too good" algorithms.

1.1.1 Roadmap to the rest of this course

Often, when we try to solve a computational problem, whether it is solving a system of linear equations, finding the top eigenvector of a matrix, or trying to rank Internet search results, it is enough to use the "I know it when I see it" standard for describing algorithms. As long as we find some way to solve the problem, we are happy and don't care so much about formal descriptions of the algorithm. But when we want to answer a question such as "does there *exist* an algorithm to solve the problem P?" we need to be much more precise.

In particular, we will need to **(1)** define exactly what does it mean to solve P, and **(2)** define exactly what is an algorithm. Even **(1)** can sometimes be non-trivial but **(2)** is particularly challenging; it is not at all clear how (and even whether) we can encompass all potential ways to design algorithms. We will consider several simple *models of computation*, and argue that, despite their simplicity, they do capture all "reasonable" approaches for computing, including all those that are currently used in modern computing devices.

Once we have these formal models of computation, we can try to obtain *impossibility results* for computational tasks, showing that some problems *can not be solved* (or perhaps can not be solved within the resources of our universe). Archimedes once said that given a fulcrum and a long enough lever, he could move the world. We will see how *reductions* allow us to leverage one hardness result into a slew of a great many others, illuminating the boundaries between the computable and uncomputable (or tractable and intractable) problems.

Later in this course we will go back to examining our models of computation, and see how resources such as randomness or quantum entanglement could potentially change the power of our model. In the context of probabilistic algorithms, we will see a glimpse of how randomness has become an indispensable tool for understanding computation, information, and communication.

We will also see how computational difficulty can be an asset rather than a hindrance, and be used for the "derandomization" of

probabilistic algorithms. The same ideas also show up in *cryptography*, which has undergone not just a technological but also an intellectual revolution in the last few decades, much of it building on the foundations that we explore in this course.

Theoretical Computer Science is a vast topic, branching out and touching upon many scientific and engineering disciplines. This course only provides a very partial (and biased) sample of this area. More than anything, I hope I will manage to "infect" you with at least some of my love for this field, which is inspired and enriched by the connection to practice, but which I find to be deep and beautiful regardless of applications.

1.2 *Exercises*

Exercise 1.1 Rank the significance of the following inventions in speeding up multiplication of large (that is 100 digit or more) numbers. That is, use "back of the envelope" estimates to order them in terms of the speedup factor they offered over the previous state of affairs.

a. Discovery of the gradeschool style digit by digit algorithm (improving upon repeated addition)

b. Discovery of Karatsuba's algorithm (improving upon the digit by digit algorithm)

c. Invention of modern electronic computers (improving upon calculations with pen and paper) ■

Exercise 1.2 The 1977 Apple II personal computer had a processor speed of 1.023 Mhz or about 10^6 operations per seconds. At the time of this writing the world's fastest supercomputer performs 93 "petaflops" (10^{15} floating point operations per second) or about 10^{18} basic steps per second. For each one of the following running times (as a function of the input length n), compute for both computers how large an input they could handle in a week of computation, if they run an algorithm that has this running time:

a. n operations.

b. n^2 operations.

c. $n \log n$ operations.

d. 2^n operations.

e. $n!$ operations. ■

Exercise 1.3 — Analysis of Karatsuba's Algorithm. a. Suppose that $T_1, T_2, T_3, \ldots$ is a sequence of numbers such that $T_2 \leq 10$ and for every n, $T_n \leq 3T_{\lceil n/2 \rceil} + Cn$. Prove that $T_n \leq 10Cn^{\log_2 3}$ for every n.[12]

b. Prove that the number of single digit operations that Karatsuba's algorithm takes to multiply two n digit numbers is at most $1000n^{\log_2 3}$. ■

[12] **Hint:** Use a proof by induction - suppose that this is true for all n's from 1 to m, prove that this is true also for $m+1$.

Exercise 1.4 Implement in the programming language of your choice functions `Gradeschool_multiply(x,y)` and `Karatsuba_multiply(x,y)` that take two arrays of digits `x` and `y` and return an array representing the product of `x` and `y` (where `x` is identified with the number `x[0]+10*x[1]+100*x[2]+...` etc..) using the gradeschool algorithm and the Karatsuba algorithm respectively. At what number of digits does the Karatsuba algorithm beat the gradeschool one? ■

1.3 Bibliographical notes

For an overview of what we'll see in this course, you could do far worse than read Bernard Chazelle's wonderful essay on the Algorithm as an Idiom of modern science.

1.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

- The *Fourier transform*, the *Fast* Fourier transform algorithm and how to use it multiply polynomials and integers. This lecture of Jeff Erickson (taken from his collection of notes) is a very good starting point. See also this MIT lecture and this popular article.
- Fast matrix multiplication algorithms, and the approach of obtaining exponent two via group representations.
- The proofs of some of the classical impossibility results in mathematics we mentioned, including the impossibility of proving Euclid's fifth postulate from the other four, impossibility of trisecting an angle with a straightedge and compass and the impossibility of solving a quintic equation via radicals. A geometric proof of the impossibility of angle trisection (one of the three geometric problems of antiquity, going back to the ancient greeks) is given in

this blog post of Tao. This book of Mario Livio covers some of the background and ideas behind these impossibility results.

1.5 *Acknowledgements*